Peer-to-Peer and Fault-Tolerance: Towards Deployment Based Technical Services

Denis Caromel, Christian Delbé, Alexandre di Costanzo

INRIA Sophia - I3S - CNRS - Université de Nice Sophia Antipolis

Abstract. For effective components, non-functional aspects must be added to the application functional code. Likewise enterprise middleware and component platforms, in the context of Grids, services must be deployed at execution in the component containers in order to implement those aspects.
This paper proposes an architecture for defining, configuring, and deploying such *Technical Services* in a Grid platform.

1 Introduction

The last decade has seen a clear identification of the so called *Non-Functional* aspects for building flexible and adaptable software. In the framework of middleware, e.g. business frameworks such as the EJB [1], architects have been making a strong point at separating the application operations, the *functional aspects*, with services that are rather orthogonal to it: transaction, persistence, security, distribution, etc.

The frameworks, such as EJB containers (JBoss, JOnAs, etc.), can further be configured for enabling and configuring such non-functional aspects. Hence, the application logic is subject to various setting and configuration, depending of the context. Overall, it opens the way to effective component codes usable in various contexts, with the crucial feature of parameterizations: choosing at deployment time various *Technical Services* to be added to the application code. In the framework of Grids, current platforms are falling short to provide such flexibility. One cannot really add and configure Fault-Tolerance, Security, Load Balancing, etc. without intensive modification of the source code. Moreover, there are no coupling with the deployment infrastructures.

In an attempt to solve this shortfall of current Grid middlewares, this article proposes a framework for defining such Technical Services dynamically, based on the application needs, potentially taking into account the underlying characteristics of the infrastructure.

Section 2 introduces the ProActive middleware. An overview of the programming model based on active objects, asynchronous communications, and futures is first given, followed with a detailed presentation of the deployment model. The later relies on Virtual Nodes and XML deployment descriptors. For the sake of reasoning in the context of real technical services, Section 3 presents the Peer-to-Peer infrastructure available in ProActive, and Section 4 the Fault-Tolerance mechanism. Finally, Section 5 proposes a flexible architecture for specifying Technical Services dynamically at deployment. Taking Peer-to-Peer and Fault-Tolerance as basic example, it is showed how to appropriately combine and configures the two together.

2 ProActive a Grid Middleware

ProActive is a 100% Java library for concurrent, distributed and mobile computing originally implemented on top of RMI [2] as the transport layer, now HTTP, RMI/SSH, and Ibis are also usable as transport layer. Besides RMI services, ProActive features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronisation mechanisms, and migration of active objects with pending calls. As ProActive is built on top of standard Java APIs, it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified Java Virtual Machine (JVM).

2.1 Base Model

A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls (see Figure 1) sent to active objects are asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [3]. There is a short rendezvous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee (on Figure 1, step 1 blocks until step 2 has completed). The ProActive library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call.

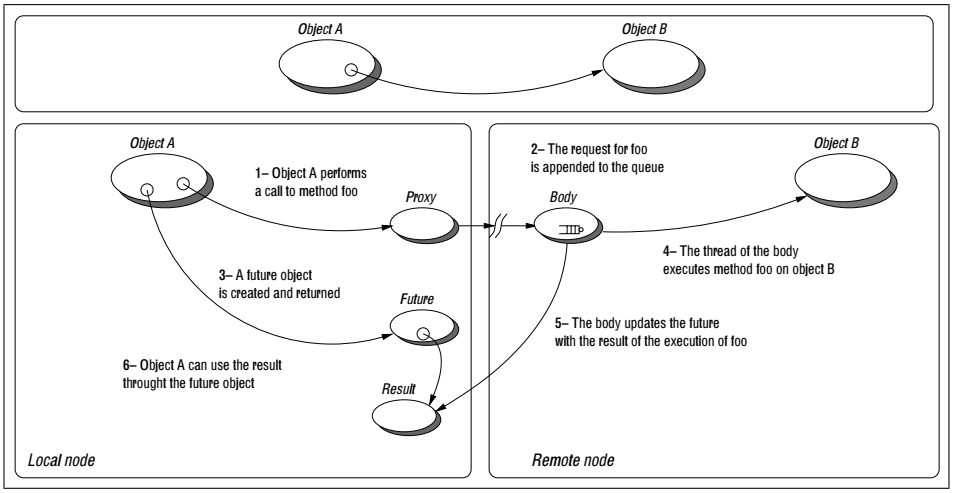

Fig. 1. Execution of a remote method call

2.2 Mapping active objects to JVMs: Nodes

Another extra feature provided by ProActive (compared to RMI for instance) is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify JVMs. *Nodes* provide those extra capabilities : a *Node* is an object defined in ProActive whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several Nodes.The traditional way to name and handle Nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance: `rmi://lo.inria.fr/Node1`.

As an active object is actually created on a *Node* we have instruction like `a = (A) ProActive.newActive("A", params, "rmi://lo.inria.fr/Node1")`.

Note that an active object can also be bound dynamically to a Node as the result of a migration.

2.3 Descriptor-based Deployment of Object-Oriented Grid Applications

The deployment of grid applications is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers and clusters. The commoditization of resources through grids and the increasing complexity of applications are making the task of deploying central and harder to perform.

ProActive succeed in completely avoid scripting for configuration, getting computing resources, etc. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code such as to gain in flexibility [4].

Principles A first principle is to *fully* eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols,

the goal being to deploy any application anywhere without changing the source code. For instance, we must be able to use various protocols, `rsh, ssh, Globus, LSF,` etc., for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as `RMIregistry, Jini, Globus, LDAP, UDDI,` etc. Therefore, we see that the creation, registration and discovery of resources has to be done externally to the application.

A second key principle is the capability to abstractly describe an application, or part of it, in term of its conceptual activities. The description should indicate the various parallel or distributed entities in the program or in the component. As we are in a (object-oriented) message passing model, to some extend, this description indicates the maximum number of address spaces. For instance, an application that is designed to use three interactive visualization Nodes, a Node to capture input from a physic

experiment, and a simulation engine designed to run on a cluster of machines should somewhere clearly advertise this information.

To summarize, in order to abstract away the underlying execution plateform, and to allow a *source-independent deployment*, a framework has to provide the following elements:

- an abstract description of the distributed entities of a parallel program or component,
- an external mapping of those entities to real *machines*, using actual *creation, registry*, and *lookup* protocols.

XML deployment descritpors To eliminate as much as possible the use of scripting languages, that can sometimes become even more complex than application code, ProActive deployement relies on XML descriptors. Indeed, the complexity is reduce thanks to XML editor tools, interactive ad-hoc environments to produce, compose, and activate descriptors. More specifically, the system relies on a specific notion, *Virtual Nodes* (VNs):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,
- a VN is defined and configured in a descriptor file (XML),
- a VN, after activation, is mapped to one or to a set of *actual ProActive Nodes*.

Of course, distributed entities (active objects), are created on Nodes, not on Virtual Nodes. There is a strong need for both Nodes and Virtual Nodes. Virtual Nodes are a much richer abstraction, as they provide mechanisms such as *set* or *cyclic mapping*. Another key aspect is the capability to describe and trigger the mapping of a single VN that generates the allocation of several JVMs. This is critical if we want to get at once machines from a cluster of PCs managed through Globus or LSF, or even more critical in a Grid application, the co-allocation of machines from several clusters across several continents.

Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between Virtual Nodes and Nodes: the function created by the deployment, the mapping. This mapping can be specified in an XML descriptor. By definition, the following operations can be configured in such a descriptor:

- the mapping of VNs to Nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup VNs.

Descriptor Example The first part of the descriptor defines the Virtual Nodes that are used in the program. Some properties can also be specified: the property `multiple` state that a Virtual Node can be mapped on several Nodes, while `unique` force the mapping on a unique Node.

```
<virtualNodesDefinition>
  <virtualNode name="vn1" property="multiple"/>
  <virtualNode name="vn2" property="unique"/>
</virtualNodesDefinition>
```

The `deployment` part allows to associate real resources and virtual Nodes. Here, the Virtual Node `vn1` is mapped onto three JVMs (`<vmName>`), and the virutal Node `vn2` is mapped on only one JVM. A Node is the result of this mapping.

```
<mapping>
  <map virtualNode="vn1">
    <jvmSet>
      <vmName value="Jvm1"/>
      <vmName value="Jvm2"/>
      <vmName value="Jvm3"/>
    </jvmSet>
  </map>
  <map virtualNode="vn2">
    <jvmSet>
      <vmName value="Jvm1"/>
    </jvmSet>
  </map>
</mapping>
```

In the next part, Nodes are associated to real processes. Those processes are started when the deployment descriptor is activated. In this example, the JVM `jvm1` will be created using the process called `localJVM`, and the `jvm2` will be created using the process called `ssh_cluster`.

```
<jvms>
  <jvm name="Jvm1">
    <creation>
      <processReference refid="localJVM"/>
    </creation>
  </jvm>
  <jvm name="Jvm2">
    <creation>
      <processReference refid="ssh_cluster"/>
    </creation>
  </jvm>
</jvms>
```

The processes are defined in the last part of the descriptor. The process `localJVM` creates a JVM on the local host. The local creation is defined in the class `process.JVMNodeProcess`.

```
<processDefinition id="localJVM">
  <jvmProcess class="process.JVMNodeProcess"/>
</processDefinition>
```

The process `ssh_cluster` creates a JVM on the remote host `Node1.cluster.inria.fr` using a ssh connexion; once connected on the remote host, a JVM is locally created using the process `localJVM`. The remote creation using ssh is defined in the class `process.ssh.SSHProcess`.

```
<processDefinition id="ssh_cluster">
  <sshProcess class="process.ssh.SSHProcess"
  hostname="Node1.cluster.inria.fr">
    <processReference refid="localJVM"/>
  </sshProcess>
</processDefinition>
```

3 Peer-to-Peer Infrastructure with ProActive

3.1 Description

Our Peer-to-Peer (P2P) infrastructure provides large scale grids for computations that would take months for achieving; these grids are a mix of clusters and desktop workstations. The P2P infrastructure can be describe by three points. The first one is that the infrastructure *does not need a central point* because it is completely *self-organized*. The second point is that is *flexible*. The last one is that the infrastructure is composed of dynamic JVMs, which run on cluster nodes and on desktop workstations.

Before going on to consider the P2P infrastructure, it is important to define what P2P is. There are many P2P definitions, many of them are similar to other distributed infrastructures, such as grid, client / server, etc. For us the P2P definition is the one defined by [5] as "Pure Peer-to-Peer Network", this definition focus on sharing, decentralization, instability, and fault tolerance.

Managing different sort of resources (clusters and desktop workstations) as a single network of resources with a high instability of them requires a fully decentralized and dynamic approach. Therefore, mimicking data P2P networks is a good solution for sharing computational resources, where JVMs are the shared resources. Because the P2P infrastructure is part of ProActive, the infrastructure does not share directly JVMs but shares ProActive Nodes.

The main characteristic of the infrastructure is the peers high volatility. This is why the infrastructure aims at maintaining a created JVMs network alive while available peer exists, this is called *self-organizing*. When it is not possible to have exterior entities, such as centralized servers, which maintain peer databases, all peers should be able of staying in the infrastructure by their own means. A widely used strategy for achieving self-organization consists on maintaining, for each peer, a list of its *acquaintances*.

This idea was selected to keep the infrastructure up. All peers have to maintain a list of acquaintances. At the beginning, when a fresh peer joins the network, it only knows peers from a list of potential network members. Because not all peers are always available, knowing a fixed number of acquaintances is a problem for peers to stay connected in the infrastructure.

Therefore, the infrastructure uses a specific parameter called *Number of Acquaintances* (NOA): the minimum number of known acquaintances for each peer. Peers update their acquaintance list every *Time to Update* (TTU), checking their own acquaintances lists to remove unavailable peers, and if the longer of the list is less than NOA, discover new peers. To discover new acquaintances, peers send exploring messages through the infrastructure. Availability is verified by sending a *heartbeat* to the acquaintances, which is sent every TTU. These two parameters, NOA and TTU, are both configurable by the administrator, which has deployed the P2P infrastructure.

As previously said, the main goal of this P2P infrastructure is to provide a structure for sharing Nodes. The resource query mechanism used is similar to the Gnutella [6] communication system, which is based on the Breadth-First Search algorithm (BFS). The system is message-based with application-level routing. Messages are forwarded to each acquaintance, and if the message has already been received, it is dropped. The number of hops that a message can take is limited with a *Time-To-Live* (TTL) parameter. The TTL is also configurable, as NOA and TTU.

The Gnutella BFS algorithm received many justified critics [7] on scalability and bandwidth usage. ProActive asynchronous method calls with future objects, provides an enhancement to the basic BFS. Before forwarding a message, a peer, which has a free Node, waits for an acknowledgment for its Node from the Nodes requester. After an expired timeout or a non-acknowledgment, the peer does not forward the message. However, the message is forwarded until the end of TTL or until the number of asked Nodes reaches zero. The message contains the initial number of requested Nodes, decreasing its value each time a peer shares its Node. For peers, which are occupied, the message is forwarded as normal BFS.

A permanent desktop grid managed by our P2P infrastructure has been deployed on the 250 desktop workstations of the INRIA Sophia. With this desktop grid, we are the first, at our knowledge, to solve the NQueens problem with 25 queens. The experimentation took six months for solving this problem instance.

Finally, there is also a work on idleness detection and prediction for improving resource usage in a P2P systems [8] has based its study on our P2P infrastructure.

3.2 Providing Nodes to the Application

The P2P infrastructure is implemented with ProActive. Thus the shared resource is not JVMs but ProActive Nodes.

The P2P infrastructure is implemented as a ProActive application because we choose to be over communication protocols, such as RMI, HTTP, Ibis, etc. Therefore, the P2P infrastructure can use RMI or HTTP as communication layer. Hence, the P2P infrastructure is implemented with classic ProActive active objects and especially with ProActive typed group [9] for broadcasting communications between peers due to your inspired BFS.

Using active objects for the implementation is a good mapping with the idea of a peer, which is an independent entities that works as a server with a FIFO request queue. The peer is also a client which sends requests to other peers. The main active object is *P2PService*, which serves all register requests or resource queries, such as Nodes or acquaintances. The figure 2 shows a peer implementation.

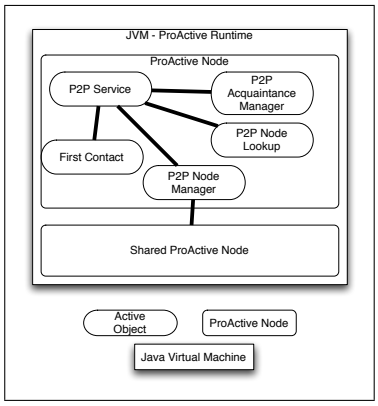

Fig. 2. A peer implementation: the peer is a JVM within two ProActive Nodes. The first Node contains all peer functional active objects and the second one is waiting for computation, i.e. it is the shared resource. The active objects roles are: *P2P Service* handles all requests with others peers; *First Contact* bootstraps the peer in the P2P network; *P2P Node Manager* manages the sharing; *P2P Acquaintance Manager* keeps the acquaintance list up-to-date, sends heartbeat, and tries to discover new peers when needed; *P2P Node Lookup* is only activated when an application asks for acquiring Nodes to the peer.

The sharing Node mechanism is an independent activity from the P2P service. This activity is handled by the *P2P Node Manager* active object (PNM). At the initialization of the PNM and of the P2P Service, the PNM instantiates the shared resource. By default, it is one Nodes by CPUs in the same JVM. An another way is to share Nodes from an XML deployment descriptor file by specifing the descriptor to the PNM, which actives the deployment and gets Nodes ready to share. When the P2P service receives a Node request, the request is forwarded to the PNM which checks for a free Node. In the case of at least one free Node, the PNM must book the Node and send back a reference to the the Node to the original request sender. However, the booking remains valid for a timeout or for an acknowledgment or none-acknowledgment from the requester.

The asking Node mechanism is allowed by the *P2P Node Lookup* (PNL), this object is activated by the P2P Service when it receives an asking Node request from an application. The PNL works as a Node broker for thew application. The PNL aims to find the number of Nodes requested by the application. It uses the BFS to frequently flood the network until it gets all Nodes or until the timeout is reached. However, the application can ask to the maximum number of Nodes, in that case the PNL asks to Nodes until the end of the application.

Finally, the asking Nodes mechanism is fully integrated to the ProActive XML deployment descriptor. This mechanism allows to share Nodes, which are already created in the P2P infrastructure. The next extract part of an XML Deployment Descriptor shows a Virtual Nodes, which acquires Nodes by a P2P infrastructure:

```
...
<virtualNodesDefinition>
  <virtualNode name="p2pSlaves" property="multiple"/>
</virtualNodesDefinition>
<mapping>
  <map virtualNode="p2pSlaves">
    <jvmSet>
      <vnName value="slaves"/>
    </jvmSet>
</mapping>
...
<jvm name="slaves">
  <acquisition>
    <aquisitionReference refid="p2pLookup"/>
  </acquisition>
</jvm>
...
<infrastructure>
 <acquisition>
  <acquisitionDefinition id="p2pLookup">
   <P2PService NodesAsked="MAX">
    <peerSet>
     <peer>rmi://registry1:3000</peer>
     <peer>rmi://registry2:3000</peer>
    </peerSet>
   </P2PService>
  </acquisitionDefinition>
 </acquisition>
</infrastructure>
...
```

We have introduced a new tag for the XML descriptor, which is `acquisition`. This tag allows to acquire Nodes for a Virtual Node from a P2P infrastructure. The parameter `NodesAsked` of the tag `P2PService` is the number of Nodes asked to the P2P infrastructure, this number can take a special value `MAX` for asking the maximum of available Nodes in the P2P infrastructure. The tag `peerSet` contains the list of peers, which are already in the P2P infrastructure.

4 Fault-Tolerance in ProActive

As the use of desktop grids goes mainstream, the need for adapted Fault-Tolerance mechanisms increases. Indeed, the probability of failure is dramatically high for such systems: a large number of resources imply a high probability of failure of one of those resources. Moreover, public Internet resources are by nature unreliable.

Rollback-recovery [10] is one solution to achieve Fault-Tolerance: the state of the application is regularly saved and stored on a stable storage. If a failure occurs, a previously recorded state is used to recover the application. Two main approaches can be distinguished : the *checkpoint-based* [11] approach, relying on recording the state of

the processes, and the *log-based* [12] approach, relying on logging and replaying inter-process messages.

Basically, we can compare those two approaches regarding two metrics: the failure-free overhead, i.e. the additional execution time induced by the Fault-Tolerance mechanism without failure, and the recovery time, i.e. the additional execution time induced by a failure during the execution. Roughly, a checkpoint-based technique provides a low failure-free overhead but a long recovery time while a log-based mechanism induces a higher failure-free overhead but a fast recovery.

Choosing one of those two approaches highly depends on the characteristics of the application and of the underlying hardware. We thus aim to provide a Fault-Tolerance mechanism that allows to choose the best approach *at deployment time*.

4.1 Fault-Tolerance in ProActive

Fault-Tolerance in ProActive is achieved by rollback-recovery. Two different mechanisms are available: a Communication-Induced Checkpointing protocol (CIC) or a Pessimistic Message Logging protocol (PML).

Those two mechanisms rely on the availability of a stable server for:

- checkpoint and logs storage: checkpoints are sent to the server by the active objects;
- failure detection: a failure detector running on the server periodically send heartbeat messages to all the active objects;
- resource service: the server is able to find new Nodes for revoering failed active objects;
- and localization service: an active object can ask to the server for the new location of a failed and recovered active object.

Communication Induced Checkpointing (CIC) [13] In a CIC fault-tolerant application, each active object have to checkpoint at least every *TTC* (Time To Checkpoint) seconds. Those checkpoints are synchronized using the application messages so as to create a *consistent* global state of the application [14]. If a failure occurs, the *entire application* must restart from such a global state, that is to say every active objects, even the non faulty, must restart from it latest checkpoint.

The failure-free overhead induced by the CIC protocol is usually low [13], as the synchronization between active objects rely only on the messages sent by the application. Of course, this overhead depends on the TTC value, set by the programmer. The TTC value depends mainly on the assessed frequency of failures. A little TTC value leads to very frequent global state creation and thus to a little rollback in the execution in case of failure. But a little TTC value leads also to a higher failure free overhead.

The counterpart is that the recovery time could be high since all the application must restart after the failure of one or more active object.

Pessimistic Message Logging (PML) Each active object in a PML fault-tolerant application have to checkpoint at least every TTC seconds. The difference with the CIC approach is that there are no need for global synchronization as with CIC protocol since

all the messages delivered to an active object are logged on a stable storage. Each checkpoint is independent: if a failure occurs, only the faulty process have to recover from its latest checkpoint.

As for CIC protocol, the TTC value impact the global failure-free overhead, but the overhead is more linked to the communication rate of the application. Regarding the CIC protocol, the PML protocol induces a higher overhead on failure-free execution. But the recovery time is lower as a single failure does not involve all the system: only the faulty has to recover.

4.2 Fault-Tolerance Configuration

Making a ProActive application fault-tolerant is automatic and transparent to the programmer; there is no need to alter the original source code of the application. The programmer can specify at deployment time if the application must be started with Fault-Tolerance, and can select the best mechanism and configuration:

- the protocol to be used (CIC or PML)
- the Time To Checkpoint value (TTC)
- the URLs of the servers

This configuration depends on the application itself, the desired response time, and the resources used to deploy the application.

The Fault-Tolerance configuration thus cannot rely on the application source code: when the application is deployed, the active object cannot contain the desired configuration. As a consequence, we have decided to put the Fault-Tolerance settings in the execution environment, *i.e. in the Nodes*. A Node contains a Fault-Tolerance configuration, and this configuration is applied to each active object created on it. When a Node is created during the mapping of a Virtual Node onto JVMs, a Fault-Tolerance configuration is associated to that Node.

This approach allows to specify the configuration at deployment time, without any alteration in the application source code. Moreover, it can rely on the deployment mechanism already provided by ProActive.

5 P2P and Fault-Tolerance: Deployment and Technical Services

The P2P infrastructure is composed of Nodes originally started by daemons, which are installed on all peers machines. After started Nodes, they are maintained by the infrastructure itself, self-organization. Those Nodes have thus a default and static configuration, chosen by the administrator at the infrastructure installation time. However, this situation does not allow the users to choose the most adapted Fault-Tolerance configuration at deployment time for their applications, as with a deployment with explicit creation of Nodes. More generally, a user must deploy its application on a pre-configured infrastructure.

We aim to provide a unique way to specify the configuration of the Nodes at deployment time for both kinds of Nodes, created or acquired by the P2P infrastructure.

5.1 Proposition: Technical Services

This section shows our proposition for a simple and unique specification for Node configuration: the *Technical Services*.

From the programmer point of view, a technical service is a class that must implements the `TechnicalService` interface of ProActive. This class defines how to configure a Node. From the deployer point of view, a technical service is a set of tuple variables-values, each of them configuring a given aspect of the application environment.

For example, for configuring the Fault-Tolerance, a `FaultToleranceService` class is provided; it defines how the configuration is applied from a Node to the active objects hosted by this Node. The deployer of the application can then configure in the deployement descriptor the Fault-Tolerance using the Technical Service XML interface.

A Technical Service is defined as a stand-alone block in the deployment descriptor. It is attached to a Virtual Node; the configuration defined by the technical service is applied to all the Nodes mapped to this Virtual Node. A single technical service can be applied to several Virtual Nodes. We introduce the `<technicalServiceDefinitions> </technicalServiceDefinitions>` in the deployment descriptors schema. Each service is defined inside a `<services></services>` as the following example:

```
<technicalServiceDefinitions>
 <service id="service1" class="services.Service1">
  <arg name="name1" value="value1"/>
  <arg name="name2" value="value2"/>
  ...
 </service>
 <service id="service2" class="services.Service2">
 ...
</technicalServiceDefinitions>
```

The `id` attribute identifies a service in the descriptor. It is used to map a service on a Virtual Node. The `class` attribute defines the implementation of the service, a class must implement the Service interface:

```
public interface TechnicalService {
   public void init(HashMap argValues);
   public void apply(Node node);
}
```

The configuration parameters of the service are specified by `arg` tag in the deployment descriptor. Those parameters are passed to the `init` method as a hashmap associating the name of a parameter as a key and its value. The `apply` method takes as parameter the Node on which the service must be applied. This method is called after the creation or acquisition of a Node, and before the Node is returned to the application.

A Virtual Node can be configured by only *one* Technical Service. Indeed, we believe that two different Technical Services, potentialy developed by two different programmers, cannot be expected to be compliant. However, combining two Technical Services

can be done at source code level, by providing a class extending `TechnicalService` that defines the correct merging of two concurrent Technical Services. A Technical Service is attached to a Virtual Node as following:

```
<virtualNodesDefinition>
 <virtualNode name="virtualNode1"
  property="multiple" serviceRefid="service1"/>
</virtualNodesDefinition>
```

5.2 Deployment with Fault-Tolerance over P2P

Figure 3 shows a complete example of a deployment descriptor based on the P2P infrastructure. We consider a *master-slaves* application, and we suppose that this application can support several synchronized masters, for example in a hierarchical way.

This descriptor defines two Virtual Nodes: one for hosting the masters and one for hosting the slaves. Each Virtual Node is configured by a technical service defining the most adapted Fault-Tolerance configuration:

- PML protocol with a short TTC value for the masters because this part of the application have to be highly reactive if a failure occurs: a failure of a master might block this entire application.
- CIC protocol with a long TTC value for the salves because the failure of a slave does not block the entire system. Moreover, the overhead induced on the salve must be low, as the computational part of the application is realized by the slaves.

6 Conclusion and Future Works

This paper has proposed a way to attach Technical Services to Virtual Nodes, mapping non-functional aspects to the containers, dynamically at deployment and execution. More investigations are needed to investigate the fit of the architecture with respect to the complexity of Grid platforms, and to the large number of technical services to be composed and deploy.

For the moment we have integrated a Load Balancing mechanism within the P2P infrastructure [15]. Unfortunately, the Load Balancing configuration is defined at the peer boot step. We are thus now working on an implementation as a Technical Service. And in the short term, we are planning to explore the combination of two Technical Services: Fault-Tolerance and Load Balancing.

```
<ProActiveDescriptor>
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="master" property="multiple"
    serviceRefid="ft-master"/>
   <virtualNode name="slaves" property="multiple"
    serviceRefid="ft-slaves"/>
  </virtualNodesDefinition>
 </componentDefinition>
 ...
 <infrastructure>
  <processes>
   <processDefinition id="localJVM">
    <jvmProcess class="JVMNodeProcess"/>
   </processDefinition>
  </processes>
  <aquisition>
   <aquisitionDefinition id="p2pservice">
    <P2PService nodesAsked="100000">
     <peerSet>
      <peer>rmi://registry1:3000</peer>
     </peerSet>
    </P2PService>
   </acquisitionDefinition>
  </services>
 </infrastructure>
 <technicalServiceDefinitions>
  <service id="ft-master" class="services.FaultTolerance">
   <arg name="proto" value="pml"/>
   <arg name="server" value="rmi://host/FTServer1"/>
   <arg name="TTC" value="60"/>
  </service>
  <service id="ft-slaves" class="services.FaultTolerance">
   <arg name="proto" value="cic"/>
   <arg name="server" value="rmi://host/FTServer2"/>
   <arg name="TTC" value="600"/>
  </service>
 </technicalServiceDefinitions>
</ProActiveDescriptor>
```

Fig. 3. An complete example: P2P and Fault-Tolerance

References

1. Sun Microsystems: Enterprise Java beans Technology (1997) http://java.sun.com/products/ejb/.
2. Sun Microsystems: Java remote method invocation specification (1998) `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf`.
3. Caromel, D.: Towards a Method of Object-Oriented Concurrent Programming. Communications of the ACM **36** (1993) 90–102
4. Baude, F., Caromel, D., Mestre, L., Huet, F., Vayssière, J.: Interactive and descriptor-based deployment of object-oriented grid applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, IEEE Computer Society (2002) 93–102
5. Schollmeier, R.: A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In IEEE, ed.: 2001 International Conference on Peer-to-Peer Computing (P2P2001), Department of Computer and Information Science Linko pings Universitet, Sweden (2001)
6. Gnutella: Gnutella peer-to-peer network (2001) http://www.gnutella.com.
7. Ritter, J.: Why Gnutella can't scale. No, really. (2001) http://www.darkridge.com/ jpr5/doc/gnutella.html.
8. Elton Nicoletti Mathias, Marcelo Veiga Neves, A.S.C., Pasin, M.: Idleness detection on distributed environments. In: Procedings of the 5th Regional School on High Performance Computing, SBC - Brazilian Computer Society (2005) 145–149
9. Baduel, L., Baude, F., Caromel, D.: Efficient, Flexible, and Typed Group Communications in Java. In: Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, ACM Press (2002) 28–36 ISBN 1-58113-559-8.
10. Elnozahy, M., Alvisi, L., Wang, Y., Johnson, D.: A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (1996)
11. Manivannan, D., Singhal, M.: Quasi-synchronous checkpointing: Models, characterization, and classification. In: IEEE Transactions on Parallel and Distributed Systems. Volume 10. (1999) 703–713
12. Alvisi, L., Marzullo, K.: Message logging: Pessimistic, optimistic, causal, and optimal. Software Engineering **24** (1998) 149–159
13. Baude, F., Caromel, D., Delbé, C., Henrio, L.: A hybrid message logging-cic protocol for constrained checkpointability. In: Proceedings of EuroPar2005. (2005)
14. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. In: ACM Transactions on Computer Systems. (1985) 63–75
15. Bustos-Jimenez, J., Caromel, D., di Costanzo, A., nd Jose M. Piquer, M.L.: Balancing active objects on a peer to peer infrastructure. In: Proceedings of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005), Valdivia, Chile, IEEE (2005)